- The `cp` (TVM codepage) is set to zero. If the smart contract wants to use another TVM codepage x, it must switch to it by using `SETCODEPAGE` x as the first instruction of its code.
- Control register `c0` (return continuation) is initialized by extraordinary continuation `ec_quit` with parameter 0. When executed, this continuation leads to a termination of TVM with exit code 0.
- Control register `c1` (alternative return continuation) is initialized by extraordinary continuation `ec_quit` with parameter 1. When invoked, it leads to a termination of TVM with exit code 1. (Notice that terminating with exit code 0 or 1 is considered a successful termination.)
- Control register `c2` (exception handler) is initialized by extraordinary continuation `ec_quit_exc`. When invoked, it takes the top integer from the stack (equal to the exception number) and terminates TVM with exit code equal to that integer. In this way, by default all exceptions terminate the smart-contract execution with exit code equal to the exception number.
- Control register `c3` (code dictionary) is initialized by the cell with the smart-contract code, similarly to the initial current continuation (`cc`).
- Control register `c4` (root of persistent data) is initialized by the persistent data of the smart contract.[35]
- Control register `c5` (root of temporary data) is initialized by a cell containing no data and one reference to a cell containing an instance of *SmartContractInfo*, which contains the smart contract balance and other useful information. The smart contract may replace the temporary data with whatever other temporary data it may require. However, the original content of the *SmartContractInfo* at the first reference of the cell held in `c5` is required by `SENDMSG` TVM primitives and other "output action" primitives of TVM.

[35]The persistent data of the smart contract need not be loaded in its entirety for this to occur. Instead the root is loaded, and TVM may load other cells by their references from the root only when they are accessed, thus providing a form of virtual memory.

- Control register `c6` (root of actions) is initialized by an empty cell. The "output action" primitives of TVM, such as `SENDMSG`, use `c6` to accumulate the list of actions (e.g., outbound messages) to be performed upon successful termination of the smart contract (cf. **4.2.7** and **4.2.8**).
- The *gas limits* `gas` $= (g_m, g_l, g_c, g_r)$ are initialized as follows:
  - The *maximal gas limit* g_m is set to the lesser of either the total Gram balance of the smart contract (after the the credit phase—i.e., combined with the value of the inbound message) divided by the current gas price, or the per-execution global gas limit.[36]
  - The *current gas limit* g_l is set to the lesser of either the Gram value of the inbound message divided by the gas price, or the global per-execution gas limit. In this way, always $g_l \leq g_m$. For inbound external messages $g_l = 0$, since they cannot carry any value.
  - The *gas credit* g_c is set to zero for inbound internal messages, and to the lesser of either g_m or a fixed small value (the default external message gas credit, a configurable parameter) for inbound external messages.
  - Finally, the *remaining gas limit* g_r is automatically initialized by $g_l + g_c$.

4.4.5. The initial stack of TVM for processing an internal message. After TVM is initialized as described in **4.4.4**, its stack is initialized by pushing the arguments to the `main()` function of the smart contract as follows:

- The Gram balance b of the smart contract (after crediting the value of the inbound message) is passed as an *Integer* amount of nanograms.
- The Gram balance b_m of inbound message m is passed as an *Integer* amount of nanograms.
- The inbound message m is passed as a cell, which contains a serialized value of type *Message* X, where X is the type of the message body.

[36]Both the global gas limit and the gas price are configurable parameters determined by the current state of the masterchain.

- The body $m_b : X$ of the inbound message, equal to the value of field `body` of m, is passed as a cell slice.
- Finally, the *function selector* s, an *Integer* normally equal to zero, is pushed into the stack.

After that, the code of the smart contract, equal to its initial value of `c3`, is executed. It selects the correct function according to s, which is expected to process the remaining arguments to the function and terminate afterwards.

4.4.6. Processing an inbound external message. An inbound external message is processed similarly to **4.4.4** and **4.4.5**, with the following modifications:

- The function selector s is set to -1, not to 0.
- The Gram balance b_m of inbound message is always 0.
- The initial current gas limit g_l is always 0. However, the initial gas credit $g_c > 0$.

The smart contract must terminate with $g_c = 0$ or $g_r \geq g_c$; otherwise, the transaction and the block containing it are invalid. Validators or collators suggesting a block candidate must never include transactions processing inbound external messages that are invalid.

4.4.7. Processing tick and tock transactions. The TVM stack for processing tick and tock transactions (cf. **4.2.4**) is initialized by pushing the following values:

- The Gram balance b of the current account in nanograms (an *Integer*).
- The 256-bit address ξ of the current account inside the masterchain, represented by an unsigned *Integer*.
- An integer equal to 0 for tick transactions and to -1 for tock transactions.
- The function selector s, equal to -2.

4.4.8. Processing split prepare transactions. For processing split prepare transactions (cf. **4.3.13**), the TVM stack is initialized by pushing the following values:

- The Gram balance b of the current account.
- A *Slice* containing *SplitMergeInfo* (cf. **4.3.13**).
- The 256-bit address ξ of the current account.
- The 256-bit address $\tilde{\xi}$ of the sibling account.
- An integer $0 \leq d \leq 63$, equal to the position of the only bit in which ξ and $\tilde{\xi}$ differ.
- The function selector s, equal to -3.

4.4.9. Processing merge install transactions. For processing merge install transactions (cf. **4.3.14**), the TVM stack is initialized by pushing the following values:

- The Gram balance b of the current account (already combined with the Gram balance of the sibling account).
- The Gram balance b' of the sibling account, taken from the inbound message m.
- The message m from the sibling account, automatically generated by a merge prepare transaction. Its `init` field contains the final state $\tilde{S}$ of the sibling account.
- The state $\tilde{S}$ of the sibling account, represented by a *StateInit* (cf. **3.1.7**).
- A *Slice* containing *SplitMergeInfo* (cf. **4.3.13**).
- The 256-bit address ξ of the current account.
- The 256-bit address $\tilde{\xi}$ of the sibling account.
- An integer $0 \leq d \leq 63$, equal to the position of the only bit in which ξ and $\tilde{\xi}$ differ.
- The function selector s, equal to -4.

4.4.10. Smart-contract information. The smart-contract information structure *SmartContractInfo*, passed in the first reference of the cell contained in control register `c5`, is serialized as follows:


```
smc_info#076ef1ea actions:uint16 msgs_sent:uint16
  unixtime:uint32 block_lt:uint64 trans_lt:uint64
  rand_seed:uint256 balance_remaining:CurrencyCollection
  myself:MsgAddressInt = SmartContractInfo;
```

The `rand_seed` field here is initialized deterministically starting from the `rand_seed` of the block, the account address, the hash of the inbound message being processed (if any), and the transaction logical time `trans_lt`.

4.4.11. Serialization of output actions. The *output actions* of a smart contract are accumulated in a linked list stored in control register `c6`. The list of output actions is serialized as a value of type *OutList* n, where n is the length of the list:

```
out_list_empty$_ = OutList 0;
out_list$_ {n:#} prev:^(OutList n) action:OutAction
  = OutList (n+1);
action_send_msg#0ec3c86d out_msg:^Message = OutAction;
action_set_code#ad4de08e new_code:^Cell = OutAction;
```

5 Block layout

This chapter presents the block layout used by the TON Blockchain, combining the data structures described separately in previous chapters to produce a complete description of a shardchain block. In addition to the TL-B schemes that define the representation of a shardchain block by a tree of cells, this chapter describes exact serialization formats for the resulting bags (collections) of cells, which are necessary to represent a shardchain block as a file.

Masterchain blocks are similar to shardchain blocks, but have some additional fields. The necessary modifications are discussed separately in **5.2**.

5.1 Shardchain block layout

This section lists the data structures that must be contained in a shardchain block and in the shardchain state, and concludes by presenting a formal TL-B scheme for a shardchain block.

5.1.1. Components of the shardchain state. The shardchain state consists of:

- *ShardAccounts*, the split part of the shardchain state (cf. **1.2.2**) containing the state of all accounts assigned to this shard (cf. **4.1.9**).
- *OutMsgQueue*, the output message queue of the shardchain (cf. **3.3.6**).
- *SharedLibraries*, the description of all shared libraries of the shardchain (for now, non-empty only in the masterchain).
- The logical time and the unixtime of the last modification of the state.
- The total balance of the shard.
- A hash reference to the most recent masterchain block, indirectly describing the state of the masterchain and, through it, the state of all other shardchains of the TON Blockchain (cf. **1.5.2**).

5.1.2. Components of a shardchain block. A shardchain block must contain:

- A list of *validator signatures* (cf. **1.2.6**), which is external with respect to all other contents of the block.

- *BlockHeader*, containing general information about the block (cf. **1.2.5**)
- Hash references to the immediately preceding block or blocks of the same shardchain, and to the most recent masterchain block.
- *InMsgDescr* and *OutMsgDescr*, the inbound and outbound message descriptors (cf. **3.2.8** and **3.3.5**).
- *ShardAccountBlocks*, the collection of all transactions processed in the block (cf. **4.2.17**) along with all updates of the states of the accounts assigned to the shard. This is the *split* part of the shardchain block (cf. **1.2.2**).
- The *value flow*, describing the total value imported from the preceding blocks of the same shardchain and from inbound messages, the total value exported by outbound message, the total fees collected by validators, and the total value remaining in the shard.
- A *Merkle update* (cf. [4, 3.1]) of the shardchain state. Such a Merkle update contains the hashes of the initial and final shardchain states with respect to the block, along with all new cells of the final state that have been created while processing the block.[37]

5.1.3. Common parts of the block layout for all workchains. Recall that different workchains may define their own rules for processing messages, other types of transactions, other components of the state, and other ways to serialize all this data. However, some components of the block and its state must be common for all workchains in order to maintain the interoperability between different workchains. Such common components include:

- *OutMsgQueue*, the outbound message queue of a shardchain, which is scanned by neighboring shardchains for messages addressed to them.
- The outer structure of *InMsgDescr* as a hashmap with 256-bit keys equal to the hashes of the imported messages. (The inbound message descriptors themselves need not have the same structure.)

[37] In principle, an experimental version of TON Blockchain might choose to keep only the hashes of the initial and final states of the shardchain. The Merkle update increases the block size, but it is handy for full nodes that want to keep and update their copy of the shardchain state. Otherwise, the full nodes would have to repeat all the computations contained in a block to compute the updated state of the shardchain by themselves.

- Some fields in the block header identifying the shardchain and the block, along with the paths from the block header to the other information indicated in this list.
- The value flow information.

5.1.4. TL-B scheme for the shardchain state. The shardchain state (cf. **1.2.1** and **5.1.1**) is serialized according to the following TL-B scheme:

```
ext_blk_ref$_ start_lt:uint64 end_lt:uint64
  seq_no:uint32 hash:uint256 = ExtBlkRef;

master_info$_ master:ExtBlkRef = BlkMasterInfo;

shard_ident$00 shard_pfx_bits:(## 6)
  workchain_id:int32 shard_prefix:uint64 = ShardIdent;

shard_state shard_id:ShardIdent
  out_msg_queue:OutMsgQueue
  total_balance:CurrencyCollection
  total_validator_fees:CurrencyCollection
  accounts:ShardAccounts
  libraries:(HashmapE 256 LibDescr)
  master_ref:(Maybe BlkMasterInfo)
  custom:(Maybe ^McStateExtra)
  = ShardState;
```

The field `custom` is usually present only in the masterchain and contains all the masterchain-specific data. However, other workchains may use the same cell reference to refer to their specific state data.

5.1.5. Shared libraries description. Shared libraries currently can be present only in masterchain blocks. They are described by an instance of $HashmapE(256, LibDescr)$, where the 256-bit key is the representation hash of the library, and *LibDescr* describes one library:

```
shared_lib_descr$00 lib:^Cell publishers:(Hashmap 256 False)
  = LibDescr;
```

Here `publishers` is a hashmap with keys equal to the addresses of all accounts that have published the corresponding shared library. The shared

library is preserved as long as at least one account keeps it in its published libraries collection.

5.1.6. TL-B scheme for an unsigned shardchain block. The precise format of an *unsigned* (cf. **1.2.6**) shardchain block is given by the following TL-B scheme:

```
block_info version:uint32
  not_master:(## 1)
  after_merge:(## 1) before_split:(## 1) flags:(## 13)
  seq_no:# vert_seq_no:#
  shard:ShardIdent gen_utime:uint32
  start_lt:uint64 end_lt:uint64
  master_ref:not_master?^BlkMasterInfo
  prev_ref:seq_no?^(BlkPrevInfo after_merge)
  prev_vert_ref:vert_seq_no?^(BlkPrevInfo 0)
  = BlockInfo;

prev_blk_info {merged:#} prev:ExtBlkRef
  prev_alt:merged?ExtBlkRef = BlkPrevInf merged;

unsigned_block info:^BlockInfo value_flow:^ValueFlow
  state_update:^(MERKLE_UPDATE ShardState)
  extra:^BlockExtra = Block;

block_extra in_msg_descr:^InMsgDescr
  out_msg_descr:^OutMsgDescr
  account_blocks:ShardAccountBlocks
  rand_seed:uint256
  custom:(Maybe ^McBlockExtra) = BlockExtra;
```

The field `custom` is usually present only in the masterchain and contains all the masterchain-specific data. However, other workchains may use the same cell reference to refer to their specific block data.

5.1.7. Description of total value flow through a block. The total value flow through a block is serialized according to the following TL-B scheme:

```
value_flow _:^[ from_prev_blk:CurrencyCollection
  to_next_blk:CurrencyCollection
```

```
  imported:CurrencyCollection
  exported:CurrencyCollection ]
  fees_collected:CurrencyCollection
  _:^[
  fees_imported:CurrencyCollection
  created:CurrencyCollection
  minted:CurrencyCollection
  ] = ValueFlow;
```

Recall that `_:^[...]` is a TL-B construction indicating that a group of fields has been moved into a separate cell. The last three fields may be non-zero only in masterchain blocks.

5.1.8. Signed shardchain block. A signed shardchain block is just an unsigned block augmented by a collection of validator signatures:

```
ed25519_signature#5 R:uint256 s:uint256 = CryptoSignature;

signed_block block:^Block blk_serialize_hash:uint256
  signatures:(HashmapE 64 CryptoSignature)
  = SignedBlock;
```

The *serialization hash* `blk_serialize_hash` of the unsigned block `block` is essentially a hash of a specific serialization of the block into an octet string (cf. **5.3.12** for a more detailed explanation). The signatures collected in `signatures` are Ed25519-signatures (cf. **A.3**) made with a validator's private keys of the SHA256 of the concatenation of the 256-bit representation hash of the block `block` and of its 256-bit serialization hash `blk_serialize_hash`. The 64-bit keys in dictionary `signatures` represent the first 64 bits of the public keys of the corresponding validators.

5.1.9. Serialization of a signed block. The overall procedure of serializing and signing a block may be described as follows:

1. An unsigned block B is generated, transformed into a complete bag of cells (cf. **5.3.2**), and serialized into an octet string S_B.

2. Validators sign the 256-bit combined hash

$$H_B := \text{SHA256}\big(\text{HASH}_\infty(B).\,\text{HASH}_M(S_B)\big) \tag{18}$$

   of the representation hash of B and of the Merkle hash of its serialization S_B.

3. A signed shardchain block $\tilde{B}$ is generated from B and these validator signatures as described above (cf. **5.1.8**).

4. This signed block $\tilde{B}$ is transformed into an incomplete bag of cells, which contains only the validator signatures, but the unsigned block itself is absent from this bag of cells, being its only absent cell.

5. This incomplete bag of cells is serialized, and its serialization is prepended to the previously constructed serialization of the unsigned block.

The result is the serialization of the signed block into an octet string. It may be propagated by network or stored into a disk file.

5.2 Masterchain block layout

Masterchain blocks are very similar to shardchain blocks of the basic workchain. This section lists some of the modifications needed to obtain the description of a masterchain block from the description of a shardchain block given in **5.1**.

5.2.1. Additional components present in the masterchain state. In addition to the components listed in **5.1.1**, the masterchain state must contain:

- *ShardHashes* — Describes the current shard configuration, and contains the hashes of the latest blocks of the corresponding shardchains.
- *ShardFees* — Describes the total fees collected by the validators of each shardchain.
- *ShardSplitMerge* — Describes future shard split/merge events. It is serialized as a part of *ShardHashes*.
- *ConfigParams* — Describes the values of all configurable parameters of the TON Blockchain.

5.2.2. Additional components present in masterchain blocks. In addition to the components listed in **5.1.2**, each masterchain block must contain:

- *ShardHashes* — Describes the current shard configuration, and contains the hashes of the latest blocks of the corresponding shardchains. (Notice that this component is also present in the masterchain state.)

5.2.3. Description of *ShardHashes*. *ShardHashes* is represented by a dictionary with 32-bit *workchain_ids* as keys, and "shard binary trees", represented by TL-B type *BinTree ShardDescr*, as values. Each leaf of this shard binary tree contains a value of type *ShardDescr*, which describes a single shard by indicating the sequence number `seq_no`, the logical time `lt`, and the hash `hash` of the latest (signed) block of the corresponding shardchain.

```
bt_leaf$0 {X:Type} leaf:X = BinTree X;
bt_fork$1 {X:Type} left:^(BinTree X) right:^(BinTree X)
          = BinTree X;

fsm_none$0 = FutureSplitMerge;
fsm_split$10 mc_seqno:uint32 = FutureSplitMerge;
fsm_merge$11 mc_seqno:uint32 = FutureSplitMerge;

shard_descr$_ seq_no:uint32 lt:uint64 hash:uint256
  split_merge_at:FutureSplitMerge = ShardDescr;

_ (HashmapE 32 ^(BinTree ShardDescr)) = ShardHashes;
```

Fields `mc_seqno` of `fsm_split` and `fsm_merge` are used to signal future shard merge or split events. Shardchain blocks referring to masterchain blocks with sequence numbers up to, but not including, the one indicated in `mc_seqno` are generated in the usual way. Once the indicated sequence number is reached, a shard merge or split event must occur.

Notice that the masterchain itself is omitted from *ShardHashes* (i.e., 32-bit index -1 is absent from this dictionary).

5.2.4. Description of *ShardFees*. *ShardFees* is a masterchain structure used to reflect the total fees collected so far by the validators of a shardchain. The total fees reflected in this structure are accumulated in the masterchain by crediting them to a special account, whose address is a configurable parameter. Typically this account is the smart contract that computes and distributes the rewards to all validators.

```
bta_leaf$0 {X:Type} {Y:Type} leaf:X extra:Y = BinTreeAug X Y;
```

```
bta_fork$1 {X:Type} left:^(BinTreeAug X Y)
           right:^(BinTreeAug X Y) extra:Y = BinTreeAug X Y;

_ (HashmapAugE 32 ^(BinTreeAug True CurrencyCollection)
  CurrencyCollection) = ShardFees;
```

The structure of *ShardFees* is similar to that of *ShardHashes* (cf. **5.2.3**), but the dictionary and binary trees involved are augmented by currency values, equal to the `total_validator_fees` values of the final states of the corresponding shardchain blocks. The value aggregated at the root of *ShardFees* is added together with the `total_validator_fees` of the masterchain state, yielding the total TON Blockchain validator fees. The increase of the value aggregated at the root of *ShardFees* from the initial to the final state of a masterchain block is reflected in the `fees_imported` in the value flow of that masterchain block.

5.2.5. Description of *ConfigParams*. Recall that the *configurable parameters* or the *configuration dictionary* is a dictionary `config` with 32-bit keys kept inside the first cell reference of the persistent data of the configuration smart contract γ (cf. **1.6**). The address γ of the configuration smart contract and a copy of the configuration dictionary are duplicated in fields `config_addr` and `config` of a *ConfigParams* structure, explicitly included into masterchain state to facilitate access to the current values of the configurable parameters (cf. **1.6.3**):

```
_ config_addr:uint256 config:^(Hashmap 32 ^Cell)
  = ConfigParams;
```

5.2.6. Masterchain state data. The data specific to the masterchain state is collected into *McStateExtra*, already mentioned in **5.1.4**:

```
masterchain_state_extra#cc1f
  shard_hashes:ShardHashes
  shard_fees:ShardFees
  config:ConfigParams
= McStateExtra;
```

5.2.7. Masterchain block data. Similarly, the data specific to the masterchain blocks is collected into *McBlockExtra*:

```
masterchain_block_extra#cc9f
  shard_hashes:ShardHashes
= McBlockExtra;
```

5.3 Serialization of a bag of cells

The description provided in the previous section defines the way a shardchain block is represented as a tree of cells. However, this tree of cells needs to be serialized into a file, suitable for disk storage or network transfer. This section discusses the standard ways of serializing a tree, a DAG, or a bag of cells into an octet string.

5.3.1. Transforming a tree of cells into a bag of cells. Recall that values of arbitrary (dependent) algebraic data types are represented in the TON Blockchain by *trees of cells*. Such a tree of cells is transformed into a directed acyclic graph, or *DAG*, of cells, by identifying identical cells in the tree. After that, we might replace each of the references of each cell by the 32-byte representation hash of the cell referred to and obtain a *bag of cells*. By convention, the root of the original tree of cells is a marked element of the resulting bag of cells, so that anybody receiving this bag of cells and knowing the marked element can reconstruct the original DAG of cells, hence also the original tree of cells.

5.3.2. Complete bags of cells. Let us say that a bag of cells is *complete* if it contains all cells referred to by any of its cells. In other words, a complete bag of cells does not have any "unresolved" hash references to cells outside that bag of cells. In most cases, we need to serialize only complete bags of cells.

5.3.3. Internal references inside a bag of cells. Let us say that a reference of a cell c belonging to a bag of cells B is *internal (with respect to B)* if the cell c_i referred to by this reference belongs to B as well. Otherwise, the reference is called *external*. A bag of cells is complete if and only if all references of its constituent cells are internal.

5.3.4. Assigning indices to the cells from a bag of cells. Let $c_0, \ldots, c_{n-1}$ be the n distinct cells belonging to a bag of cells B. We can list these cells in some order, and then assign indices from 0 to $n-1$, so that cell c_i gets index i. Some options for ordering cells are:

- Order cells by their representation hash. Then $\text{Hash}(c_i) < \text{Hash}(c_j)$ whenever $i < j$.
- Topological order: if cell c_i refers to cell c_j, then $i < j$. In general, there is more than one topological order for the same bag of cells. There are two standard ways for constructing topological orders:
  - Depth-first order: apply a depth-first search to the directed acyclic graph of cells starting from its root (i.e., marked cell), and list cells in the order they are visited.
  - Breadth-first order: same as above, but applying a breadth-first search.

Notice that the topological order always assigns index 0 to the root cell of a bag of cells constructed from a tree of cells. In most cases, we opt to use a topological order, or the depth-first order if we want to be more specific.

If cells are listed in a topological order, then the verification that there are no cyclic references in a bag of cells is immediate. On the other hand, ordering cells by their representation hash simplifies the verification that there are no duplicates in a serialized bag of cells.

5.3.5. Outline of serialization process. The serialization process of a bag of cells B consisting of n cells can be outlined as follows:

1. List the cells from B in a topological order: c_0, c_1, ..., c_{n-1}. Then c_0 is the root cell of B.
2. Choose an integer s, such that $n \leq 2^s$. Represent each cell c_i by an integral number of octets in the standard way (cf. **1.1.3** or [4, 3.1.4]), but using unsigned big-endian s-bit integer j instead of hash $\text{Hash}(c_j)$ to represent internal references to cell c_j (cf. **5.3.6** below).
3. Concatenate the representations of cells c_i thus obtained in the increasing order of i.
4. Optionally, an index can be constructed that consists of $n + 1$ t-bit integer entries L_0, ..., L_n, where L_i is the total length (in octets) of the representations of cells c_j with $j \leq i$, and integer $t \geq 0$ is chosen so that $L_n \leq 2^t$.

5. The serialization of the bag of cells now consists of a magic number indicating the precise format of the serialization, followed by integers $s \geq 0$, $t \geq 0$, $n \leq 2^s$, an optional index consisting of $\lceil (n+1)t/8 \rceil$ octets, and L_n octets with the cell representations.

6. An optional CRC32 may be appended to the serialization for integrity verification purposes.

If an index is included, any cell c_i in the serialized bag of cells may be easily accessed by its index i without deserializing all other cells, or even without loading the entire serialized bag of cells in memory.

5.3.6. Serialization of one cell from a bag of cells. More precisely, each individual cell $c = c_i$ is serialized as follows, provided s is a multiple of eight (usually $s = 8$, 16, 24, or 32):

1. Two descriptor bytes d_1 and d_2 are computed similarly to [4, 3.1.4] by setting $d_1 = r + 8s + 16h + 32l$ and $d_2 = \lfloor b/8 \rfloor + \lceil b/8 \rceil$, where:

   - $0 \leq r \leq 4$ is the number of cell references present in cell c; if c is absent from the bag of cells being serialized and is represented by its hashes only, then $r = 7$.[38]
   - $0 \leq b \leq 1023$ is the number of data bits in cell c.
   - $0 \leq l \leq 3$ is the level of cell c (cf. [4, 3.1.3]).
   - $s = 1$ for exotic cells and $s = 0$ for ordinary cells.
   - $h = 1$ if the cell's hashes are explicitly included into the serialization; otherwise, $h = 0$. (When $r = 7$, we must always have $h = 1$.)

   For absent cells (i.e., external references), only d_1 is present, always equal to $23 + 32l$.

2. Two bytes d_1 and d_2 (if $r < 7$) or one byte d_1 (if $r = 7$) begin the serialization of cell c.

[38] Notice that these "absent cells" are different from the library reference and external reference cells, which are kinds of exotic cells (cf. [4, 3.1.7]). Absent cells, by contrast, are introduced only for the purpose of serializing incomplete bags of cells, and can never be processed by TVM.

3. If $h = 1$, the serialization is continued by $l + 1$ 32-byte higher hashes of c (cf. [4, 3.1.6]): $\text{Hash}_1(c)$, ..., $\text{Hash}_{l+1}(c) = \text{Hash}_\infty(c)$.

4. After that, $\lceil b/8 \rceil$ data bytes are serialized, by splitting b data bits into 8-bit groups and interpreting each group as a big-endian integer in the range $0 \ldots 255$. If b is not divisible by 8, then the data bits are first augmented by one binary `1` and up to six binary `0`, so as to make the number of data bits divisible by eight.[39]

5. Finally, r cell references to cells c_{j_1}, ..., c_{j_r} are encoded by means of r s-bit big-endian integers j_1, ..., j_r.[40]

5.3.7. A classification of serialization schemes for bags of cells. A serialization scheme for a bag of cells must specify the following parameters:

- The 4-byte magic number prepended to the serialization.
- The number of bits s used to represent cell indices. Usually s is a multiple of eight (e.g., 8, 16, 24, or 32).
- The number of bits t used to represent offsets of cell serializations (cf. **5.3.5**). Usually t is also a multiple of eight.
- A flag indicating whether an index with offsets L_0, ..., L_n of cell serializations is present. This flag may be combined with t by setting $t = 0$ when the index is absent.
- A flag indicating whether the CRC32-C of the whole serialization is appended to it for integrity verification purposes.

5.3.8. Fields present in the serialization of a bag of cells. In addition to the values listed in **5.3.7**, fixed by the choice of a serialization scheme for bags of cells, the serialization of a specific bag of cells must specify the following parameters:

- The total number of cells n present in the serialization.

[39] Notice that exotic cells (with $s = 1$) always have $b \geq 8$, with the cell type encoded in the first eight data bits (cf. [4, 3.1.7]).

[40] If the bag of cells is not complete, some of these cell references may refer to cells c' absent from the bag of cells. In that case, special "absent cells" with $r = 7$ are included into the bag of cells and are assigned some indices j. These indices are then used to represent references to absent cells.

- The number of "root cells" $k \leq n$ present in the serialization. The root cells themselves are $c_0, \ldots, c_{k-1}$. All other cells present in the bag of cells are expected to be reachable by chains of references starting from the root cells.
- The number of "absent cells" $l \leq n - k$, which represent cells that are actually absent from this bag of cells, but are referred to from it. The absent cells themselves are represented by $c_{n-l}, \ldots, c_{n-1}$, and only these cells may (and also must) have $r = 7$. Complete bags of cells have $l = 0$.
- The total length in bytes L_n of the serialization of all cells. If the index is present, L_n might not be stored explicitly since it can be recovered as the last entry of the index.

5.3.9. TL-B scheme for serializing bags of cells. Several TL-B constructors can be used to serialize bags of cells into octet (i.e., 8-bit byte) sequences:

```
serialized_boc_tiny cells:uint8 roots:uint8 absent:uint8
  tot_cells_size:(## 32) cells:(tot_cells_size * [ uint8 ])
  = BagOfCells;
serialized_boc_small cells:uint16 roots:uint16 absent:uint16
  tot_cells_size:(## 32) cells:(tot_cells_size * [ uint8 ])
  = BagOfCells;
serialized_boc_medium cells:uint24 roots:uint24 absent:uint24
  tot_cells_size:(## 64) cells:(tot_cells_size * [ uint8 ])
  = BagOfCells;
serialized_boc_large cells:uint32 roots:uint32 absent:uint32
  tot_cells_size:(## 64) cells:(tot_cells_size * [ uint8 ])
  = BagOfCells;
```

Field `cells` is n, `roots` is k, `absent` is l, and `tot_cells_size` is L_n (the total size of the serialization of all cells in bytes).

If an index is present, parameters $s/8$ and $t/8$ are serialized separately as 8-bit fields `size` and `off_bytes`, respectively:

```
serialized_boc_idx size:(## 8) { size <= 4 }
  off_bytes:(## 8) { off_bytes <= 8 }
  cells:(##(size * 8))
```

```
  roots:(##(size * 8))
  absent:(##(size * 8)) { roots + absent <= cells }
  tot_cells_size:(##(off_bytes * 8))
  index:(cells * [ ##(off_bytes * 8) ])
  cells:(tot_cells_size * [ uint8 ])
  = BagOfCells;
```

Finally, constructors `serialized_boc_*_crc32`, with the asterisk replaced by either `tiny`, `small`, `medium`, `large`, or `idx`, are also introduced, with one extra field `crc32c:uint32` added as the last field.

5.3.10. Storing compiled TVM code in files. Notice that the above procedure for serializing bags of cells may be used to serialize compiled smart contracts and other TVM code. One must define a TL-B constructor similar to the following:

```
compiled_smart_contract
  compiled_at:uint32 code:^Cell data:^Cell
  description:(Maybe ^TinyString)
  _:^[ source_file:(Maybe ^TinyString)
       compiler_version:(Maybe ^TinyString) ]
  = CompiledSmartContract;

tiny_string#_ len:(#<= 126) str:(len * [ uint8 ]) = TinyString;
```

Then a compiled smart contract may be represented by a value of type *CompiledSmartContract*, transformed into a tree of cells and then into a bag of cells, and then serialized using one of the constructors listed in **5.3.9**. The resulting octet string may be then written into a file with suffix `.tvc` ("TVM smart contract"), and this file may be used to distribute the compiled smart contract, download it into a wallet application for deploying into the TON Blockchain, and so on.

5.3.11. Merkle hashes for an octet string. On some occasions, we must define a Merkle hash $\textsc{Hash}_M(s)$ of an arbitrary octet string s of length $|s|$. We do this as follows:

- If $|s| \le 256$ octets, then the Merkle hash of s is just its SHA256:

$$\textsc{Hash}_M(s) := \textsc{sha256}(s) \quad \text{if } |s| \le 256. \tag{19}$$

- If $|s| > 256$, let $n = 2^k$ be the largest power of two less than $|s|$ (i.e., $k := \lfloor \log_2(|s|-1) \rfloor$, $n := 2^k$). If s' is the prefix of s of length n, and s'' is the suffix of s of length $|s| - n$, so that s is the concatenation $s'.s''$ of s' and s'', we define

$$\text{Hash}_M(s) := \text{sha256}\big(\text{int}_{64}(|s|).\,\text{Hash}_M(s').\,\text{Hash}_M(s'')\big) \tag{20}$$

  In other words, we concatenate the 64-bit big-endian representation of $|s|$ and the recursively computed Merkle hashes of s' and s'', and compute sha256 of the resulting string.

One can check that $\text{Hash}_M(s) = \text{Hash}_M(t)$ for octet strings s and t of length less than $2^{64} - 2^{56}$ implies $s = t$ unless a hash collision for sha256 has been found.

5.3.12. The serialization hash of a block. The construction of **5.3.11** is applied in particular to the serialization of the bag of cells representing an unsigned shardchain or masterchain block. The validators sign not only the representation hash of the unsigned block, but also the "serialization hash" of the unsigned block, defined as Hash_M of the serialization of the unsigned block. In this way, the validators certify that this octet string is indeed a serialization of the corresponding block.

References

[1] Daniel J. Bernstein, *Curve25519: New Diffie–Hellman Speed Records* (2006), in: M. Yung, Ye. Dodis, A. Kiayas et al, *Public Key Cryptography*, Lecture Notes in Computer Science **3958**, pp. 207–228. Available at `https://cr.yp.to/ecdh/curve25519-20060209.pdf`.

[2] Daniel J. Bernstein, Niels Duif, Tanja Lange et al., *High-speed high-security signatures* (2012), *Journal of Cryptographic Engineering* **2** (2), pp. 77–89. Available at `https://ed25519.cr.yp.to/ed25519-20110926.pdf`.

[3] N. Durov, *Telegram Open Network*, 2017.

[4] N. Durov, *Telegram Open Network Virtual Machine*, 2018.

A Elliptic curve cryptography

This appendix contains a formal description of the elliptic curve cryptography currently used in TON, particularly in the TON Blockchain and the TON Network.

TON uses two forms of elliptic curve cryptography: Ed25519 is used for cryptographic Schnorr signatures, while Curve25519 is used for asymmetric cryptography. These curves are used in the standard way (as defined in the original articles [1] and [2] by D. Bernstein and RFCs 7748 and 8032); however, some serialization details specific to TON must be explained. One unique adaptation of these curves for TON is that TON supports automatic conversion of Ed25519 keys into Curve25519 keys, so that the same keys can be used for signatures and for asymmetric cryptography.

A.1 Elliptic curves

Some general facts on elliptic curves over finite fields, relevant for elliptic curve cryptography, are collected in this section.

A.1.1. Finite fields. We consider elliptic curves over finite fields. For the purposes of the Curve25519 and Ed25519 algorithms, we will be mostly concerned with elliptic curves over the finite prime field $k := \mathbb{F}_p$ of residues modulo p, where $p = 2^{255} - 19$ is a prime number, and over finite extensions $\mathbb{F}_q$ of $\mathbb{F}_p$, especially the quadratic extension $\mathbb{F}_{p^2}$.[41]

A.1.2. Elliptic curves. An *elliptic curve* $E = (E, O)$ over a field k is a geometrically integral smooth projective curve E/k of genus $g = 1$, along with a marked k-rational point $O \in E(k)$. It is well-known that an elliptic curve E over a field k can be represented in (generalized) Weierstrass form:

$$y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6 \quad \text{for some } a_1, \ldots, a_6 \in k. \tag{21}$$

More precisely, this is only the affine part of the elliptic curve, written in coordinates (x, y). For any field extension K of k, $E(K)$ consists of all solutions $(x, y) \in K^2$ of equation (21), called *finite points of* $E(K)$, along with a point at infinity, which is the marked point O.

[41] Arithmetic modulo p for a modulus p near a power of two can be implemented very efficiently. On the other hand, residues modulo $2^{255} - 19$ can be represented by 255-bit integers. This is the reason this particular value of p has been chosen by D. Bernstein.

A.1.3. Weierstrass form in homogeneous coordinates. In homogeneous coordinates $[X : Y : Z]$, (21) corresponds to

$$Y^2Z + a_1XYZ + a_3YZ^2 = X^3 + a_2X^2Z + a_4XZ^2 + a_6Z^3 \tag{22}$$

When $Z \neq 0$, we can set $x := X/Z$, $y := Y/Z$, and obtain a solution (x, y) of (21) (i.e., a finite point of E). On the other hand, the only solution (up to proportionality) of (22) with $Z = 0$ is $[0 : 1 : 0]$; this is the point at infinity O.

A.1.4. Standard Weierstrass form. When the characteristic $\operatorname{char} k$ of field k is $\neq 2$, 3, the Weierstrass form of (21) or (22) can be simplified with the aid of linear transformations $y' := y + a_1x/2 + a_3/2$, $x' := x + a_2/3$, thus making $a_1 = a_3 = a_2 = 0$ and obtaining

$$y^2 = x^3 + a_4x + a_6 \tag{23}$$

and

$$Y^2Z = X^3 + a_4XZ^2 + a_6Z^3 \tag{24}$$

Such an equation defines an elliptic curve if and only if the cubic polynomial $P(x) := x^3 + a_4x + a_6$ has no multiple roots, i.e., if the discriminant $D := -4a_4^3 - 27a_6^2$ is non-zero.

A.1.5. Addition of points on elliptic curve E. Let K be a field extension of field k, and let $E = (E, O)$ be any elliptic curve in Weierstrass form defined over k. Then any line $l \subset \mathbb{P}^2_K$ intersects the elliptic curve $E_{(K)}$ (which is the base change of curve E to field K, i.e., the curve defined by the same equations over a larger field K) at exactly three points P, Q, R, considered with multiplicities. We define the *addition of points* on elliptic curve E (or rather the addition of its K-valued points $E(K)$) by requiring that

$$P + Q + R = O \quad \text{whenever } \{P, Q, R\} = l \cap E \text{ for some line } l \subset \mathbb{P}^2_K. \tag{25}$$

It is well-known that this requirement defines a unique commutative law $[+] : E \times_k E \to E$ on the points of the elliptic curve E, having O as its neutral element. When elliptic curve E is represented by its Weierstrass form (21), one can write explicit formulas expressing the coordinates x_{P+Q}, y_{P+Q} of the sum $P+Q$ of two K-valued points P, $Q \in E(K)$ of elliptic curve E as rational functions of the coordinates x_P, y_P, x_Q, $y_Q \in K$ of points P and Q and of the coefficients $a_i \in k$ of (21).

A.1.6. Power maps. Since $E(K)$ is an abelian group, one can define *multiples* or *powers* $[n]X$ for any point $X \in E(K)$ and any integer $n \in \mathbb{Z}$. If $n = 0$, then $[0]X = O$; if $n > 0$, then $[n]X = [n-1]X + X$; if $n < 0$, then $[n]X = -[-n]X$. The map $[n] = [n]_E : E \to E$ for $n \neq 0$ is an *isogeny*, meaning that it is a non-constant homomorphism for the group law of E:

$$[n](P+Q) = [n]P + [n]Q \quad \text{for any } P, Q \in E(K) \text{ and } n \in \mathbb{Z}. \tag{26}$$

In particular, $[-1]_E : E \to E$, $P \mapsto -P$, is an involutive automorphism of elliptic curve E. If E is in Weierstrass form, $[-1]_E$ maps $(x, y) \mapsto (x, -y)$, and two points $P, Q \in E(\mathbb{F}_q)$ have equal x-coordinates if and only if $Q = \pm P$.

A.1.7. The order of the group of rational points of E. Let E be an elliptic curve defined over a finite base field k, and let $K = \mathbb{F}_q$ be a finite extension of k. Then $E(\mathbb{F}_q)$ is a finite abelian group. By a well-known result of Hasse, the order n of this group is not too distant from q:

$$n = |E(\mathbb{F}_q)| = q - t + 1 \quad \text{where } t^2 \leq 4q, \text{ i.e., } |t| \leq 2\sqrt{q}. \tag{27}$$

We will be mostly interested in the case $K = k = \mathbb{F}_p$, with $q = p$ a prime number.

A.1.8. Cyclic subgroups of large prime order. Elliptic curve cryptography is usually performed using elliptic curves that admit a (necessarily cyclic) subgroup $C \subset E(\mathbb{F}_q)$ of prime order ℓ. Equivalently, a rational point $G \in E(\mathbb{F}_q)$ of prime order ℓ may be given; then C can be recovered as the cyclic subgroup $\langle G \rangle$ generated by G. In order to verify that a point $G \in E(\mathbb{F}_q)$ generates a cyclic group of prime order ℓ, one can check that $G \neq O$, but $[\ell]G = O$.

By the Legendre theorem, ℓ is necessarily a divisor of the order $n = |E(\mathbb{F}_q)|$ of the finite abelian group $E(\mathbb{F}_q)$:

$$n = |E(\mathbb{F}_q)| = c\ell \quad \text{for some integer } c \geq 1 \tag{28}$$

The integer c is called the *cofactor*; one usually wants the cofactor to be as small as possible, so as to make $\ell = n/c$ as large as possible. Recall that n always has the same order of magnitude as q by (27), so it cannot be changed much by varying E once q is fixed.

A.1.9. Data for elliptic curve cryptography. In order to define specific elliptic curve cryptography, one must fix a finite base field $\mathbb{F}_q$ (if $q = p$ is a prime, it is sufficient to fix prime p), an elliptic curve $E/\mathbb{F}_q$ (usually represented by the coefficients of its Weierstrass form (23) or (21)), the base point O (which usually is the point at infinity of an elliptic curve written in Weierstrass form), and the generator $G \in E(\mathbb{F}_q)$ (usually determined by its coordinates (x, y) with respect to the equation of the elliptic curve) of a cyclic subgroup of large prime order ℓ. Prime number ℓ and the cofactor c are usually also part of the elliptic cryptography data.

A.1.10. Main operations of elliptic curve cryptography. Elliptic curve cryptography usually deals with a fixed cyclic subgroup C of a large prime order ℓ inside the group of points of an elliptic curve E over a finite field $\mathbb{F}_q$. A generator G of C is usually fixed. It is usually assumed that, given a point X of C, one cannot find its "discrete logarithm base G" (i.e., a residue n modulo ℓ such that $X = [n]G$) faster than in $O(\sqrt{\ell})$ operations. The most important operations used in elliptic curve cryptography are the addition of points from $C \subset E(\mathbb{F}_q)$ and the computation of their powers, or multiples.

A.1.11. Private and public keys for elliptic curve cryptography. Usually a private key for elliptic curve cryptography described by the data listed in **A.1.9** is a "random" integer $0 < a < \ell$, called the *secret exponent*, and the corresponding public key is the point $A := [a]G$ (or just its x-coordinate x_A), suitably serialized.

A.1.12. Montgomery curves. Elliptic curves with the specific Weierstrass equation

$$y^2 = x^3 + Ax^2 + x \quad \text{where } A = 4a - 2 \text{ for some } a \in k,\ a \neq 0,\ a \neq 1 \tag{29}$$

are called *Montgomery curves*. They have the convenient property that $x_{P+Q}x_{P-Q}$ can be expressed as a simple rational function of x_P and x_Q:

$$x_{P+Q}x_{P-Q} = \left(\frac{x_P x_Q - A}{x_P - x_Q}\right)^2 \tag{30}$$

This means that x_{P+Q} can be computed provided x_{P-Q}, x_P, and x_Q are known. In particular, if x_P, $x_{[n]P}$, and $x_{[n+1]P}$ are known, then $x_{[2n]P}$, $x_{[2n+1]P}$, and $x_{[2n+2]P}$ can be computed. Using the binary representation of $0 < n < 2^s$, one can compute $x_{[\lfloor n/2^{s-i} \rfloor]P}$, $x_{[\lfloor n/2^{s-i} \rfloor+1]P}$ for $i = 0, 1, \ldots, s$, thus obtaining

$x_{[n]P}$ (this algorithm for quickly computing $x_{[n]P}$ starting from x_P on Montgomery curves is called a *Montgomery ladder*). Hence we see the importance of Montgomery curves for elliptic curve cryptography.

A.2 Curve25519 cryptography

This section describes the well-known Curve25519 cryptography proposed by Daniel Bernstein [1] and its usage in TON.

A.2.1. Curve25519. *Curve25519* is defined as the Montgomery curve

$$y^2 = x^3 + Ax^2 + x \quad \text{over } \mathbb{F}_p, \text{ where } p = 2^{255} - 19 \text{ and } A = 486662. \tag{31}$$

The order of this curve is 8ℓ, where ℓ is a prime number, and $c = 8$ is the cofactor. The cyclic subgroup of order ℓ is generated by a point G with $x_G = 9$ (this determines G up to the sign of y_G, which is unimportant). The order of the quadratic twist $2y^2 = x^3 + Ax^2 + x$ of Curve25519 is $4\ell'$ for another prime number ℓ'.[42]

A.2.2. Parameters of Curve25519. The parameters of Curve25519 are as follows:

- Base field: Prime finite field $\mathbb{F}_p$ for $p = 2^{255} - 19$.
- Equation: $y^2 = x^3 + Ax^2 + x$ for $A = 486662$.
- Base point G: Characterized by $x_G = 9$ (nine is the smallest positive integer x-coordinate of a generator of the subgroup of large prime order of $E(\mathbb{F}_p)$).
- Order of $E(\mathbb{F}_p)$:

$$|E(\mathbb{F}_p)| = p - t + 1 = 8\ell, \quad \text{where} \tag{32}$$

$$\ell = 2^{252} + 27742317777372353535851937790883648493 \quad \text{is prime.} \tag{33}$$

[42] Actually, D. Bernstein chose $A = 486662$ because it is the smallest positive integer $A \equiv 2 \pmod 4$ such that both the corresponding Montgomery curve (31) over $\mathbb{F}_p$ for $p = 2^{255} - 19$ and the quadratic twist of this curve have small cofactors. Such an arrangement avoids the necessity to check whether an x-coordinate $x_P \in \mathbb{F}_p$ of a point P defines a point $(x_P, y_P) \in \mathbb{F}_p^2$ lying on the Montgomery curve itself or on its quadratic twist.

- Order of $\tilde{E}(\mathbb{F}_p)$, where $\tilde{E}$ is the quadratic twist of E:

$$|\tilde{E}(\mathbb{F}_p)| = p + t + 1 = 2p + 2 - 8\ell = 4\ell', \quad \text{where} \tag{34}$$

$$\ell' = 2^{253} - 55484635554744707071703875581767296995 \quad \text{is prime.} \tag{35}$$

A.2.3. Private and public keys for standard Curve25519 cryptography. A private key for Curve25519 cryptography is usually defined as a *secret exponent* a, while the corresponding public key is x_A, the x-coordinate of point $A := [a]G$. This is usually sufficient for performing ECDH (elliptic curve Diffie–Hellman key exchange) and asymmetric elliptic curve cryptography, as follows:

If a party wants to send a message M to another party, which has public key x_A (and private key a), the following computations are performed. A one-time random secret exponent b is generated, and $x_B := x_{[b]G}$ and $x_{[b]A}$ are computed using a Montgomery ladder. After that, the message M is encrypted by a symmetric cypher such as AES using the 256-bit "shared secret" $S := x_{[b]A}$ as a key, and 256-bit integer ("one-time public key") x_B is prepended to the encrypted message. Once the party with public key x_A receives this message, it can compute $x_{[a]B}$ starting from x_B (transmitted with the encrypted message) and the private key a. Since $x_{[a]B} = x_{[ab]G} = x_{[b]A} = S$, the receiving party recovers the shared secret S and is able to decrypt the remainder of the message.

A.2.4. Public and private keys for TON Curve25519 cryptography. TON uses another form for public and private keys of Curve25519 cryptography, borrowed from Ed25519 cryptography.

A private key for TON Curve25519 cryptography is just a random 256-bit string k. It is used by computing $\text{SHA512}(k)$, taking the first 256 bits of the result, interpreting them as a little-endian 256-bit integer a, clearing bits 0, 1, 2, and 255 of a, and setting bit 254 so as to obtain a value $2^{254} \le a < 2^{255}$, divisible by eight. The value a thus obtained is the *secret exponent* corresponding to k; meanwhile, the remaining 256 bits of $\text{SHA512}(k)$ constitute the *secret salt* k''.

The public key corresponding to k—or to the secret exponent a—is just the x-coordinate x_A of the point $A := [a]G$. Once a and x_A are computed, they are used in exactly the same way as in **A.2.3**. In particular, if x_A needs to be serialized, it is serialized into 32 octets as an unsigned little-endian 256-bit integer.

A.2.5. Curve25519 is used in the TON Network. Notice that the asymmetric Curve25519 cryptography described in **A.2.4** is extensively used by the TON Network, especially the ADNL (Abstract Datagram Network Layer) protocol. However, TON Blockchain needs elliptic curve cryptography mostly for signatures. For this purpose, Ed25519 signatures described in the next section are used.

A.3 Ed25519 cryptography

Ed25519 cryptography is extensively used for fast cryptographic signatures by both the TON Blockchain and the TON Network. This section describes the variant of Ed25519 cryptography used by TON. An important difference from the standard approaches (as defined by D. Bernstein et al. in [2]) is that TON provides automatic conversion of private and public Ed25519 keys into Curve25519 keys, so that the same keys could be used both for encrypting/decrypting and for signing messages.

A.3.1. Twisted Edwards curves. A *twisted Edwards curve* $E_{a,d}$ with parameters $a \neq 0$ and $d \neq 0, a$ over a field k is given by equation

$$E_{a,d}: ax^2 + y^2 = 1 + dx^2y^2 \quad \text{over } k \tag{36}$$

If $a = 1$, this equation defines an (untwisted) Edwards curve. Point $O(0, 1)$ is usually chosen as the marked point of $E_{a,d}$.

A.3.2. Twisted Edwards curves are birationally equivalent to Montgomery curves. A twisted Edwards curve $E_{a,d}$ is birationally equivalent to a Montgomery elliptic curve

$$M_A: v^2 = u^3 + Au^2 + u \tag{37}$$

where $A = 2(a + d)/(a - d)$ and $d/a = (A - 2)/(A + 2)$. The birational equivalence $\phi : E_{a,d} \dashrightarrow M_A$ and its inverse ϕ^{-1} are given by

$$\phi : (x, y) \mapsto \left(\frac{1+y}{1-y}, \frac{c(1+y)}{x(1-y)}\right) \tag{38}$$

and

$$\phi^{-1} : (u, v) \mapsto \left(\frac{cu}{v}, \frac{u-1}{u+1}\right) \tag{39}$$

where

$$c = \sqrt{\frac{A+2}{a}} \tag{40}$$

Notice that ϕ transforms the marked point $O(0,1)$ of $E_{a,d}$ into the marked point of M_A (i.e., its point at infinity).

A.3.3. Addition of points on a twisted Edwards curve. Since $E_{a,d}$ is birationally equivalent to an elliptic curve M_A, the addition of points on M_A can be transferred to $E_{a,d}$ by setting

$$P + Q := \phi^{-1}\big(\phi(P) + \phi(Q)\big) \quad \text{for any } P,\, Q \in E_{a,d}(k). \tag{41}$$

Notice that the marked point $O(0,1)$ is the neutral element with respect to this addition, and that $-(x_P, y_P) = (-x_P, y_P)$.

A.3.4. Formulas for adding points on a twisted Edwards curve. The coordinates x_{P+Q} and y_{P+Q} admit simple expressions as rational functions of x_P, y_P, x_Q, y_Q:

$$x_{P+Q} = \frac{x_P y_Q + x_Q y_P}{1 + d x_P x_Q y_P y_Q} \tag{42}$$

$$y_{P+Q} = \frac{y_P y_Q - a x_P x_Q}{1 - d x_P x_Q y_P y_Q} \tag{43}$$

These expressions can be efficiently computed, especially if $a = -1$. This is the reason twisted Edwards curves are important for fast elliptic curve cryptography.

A.3.5. Ed25519 twisted Edwards curve. Ed25519 is the twisted Edwards curve $E_{-1,d}$ over $\mathbb{F}_p$, where $p = 2^{255} - 19$ is the same prime number as that used for Curve25519, and $d = -(A-2)/(A+2) = -121665/121666$, where $A = 486662$ is the same as in the equation (31):

$$-x^2 + y^2 = 1 - \frac{121665}{121666} x^2 y^2 \quad \text{for } x,\, y \in \mathbb{F}_p,\, p = 2^{255} - 19. \tag{44}$$

In this way, Ed25519-curve $E_{-1,d}$ is birationally equivalent to Curve25519 (31), and one can use $E_{-1,d}$ and formulas (42)–(43) for point addition on either Ed25519 or Curve25519, using (38) and (39) to convert points on Ed25519 into corresponding points on Curve25519, and vice versa.

A.3.6. Generator of Ed25519. The generator of Ed25519 is the point G' with $y(G') = 4/5$ and $0 \leq x(G') < p$ even. According to (38), it corresponds to the point (u, v) of Curve25519 with $u = (1 + 4/5)/(1 - 4/5) = 9$ (i.e., to the generator G of Curve25519 chosen in **A.2.2**). In particular, $G = \phi(G')$, G' generates a cyclic subgroup of the same large prime order ℓ given in (32), and for any integer a,

$$\phi([a]G') = [a]G \quad . \tag{45}$$

In this way, we can perform computations with Curve25519 and its generator G, or with Ed25519 and generator G', and obtain essentially the same results.

A.3.7. Standard representation of points on Ed25519. A point $P(x, y)$ on Ed25519 may be represented by its two coordinates x_P and y_P, residues modulo $p = 2^{255} - 19$. In their turn, both these coordinates may be represented by unsigned 255- or 256-bit integers $0 \leq x_P, y_P < p < 2^{255}$.

However, a more compact representation of P by one little-endian unsigned 256-bit integer $\tilde{P}$ is commonly used (and is used by TON as well). Namely, the 255 lower-order bits of $\tilde{P}$ contain y_P, $0 \leq y_P < p < 2^{255}$, and bit 255 is used to store $x_P \bmod 2$, the lower-order bit of x_P. Since y_P always determines x_P up to sign (i.e., up to replacing x_P with $p - x_P$), x_P and $p - x_P$ can always be distinguished by their lower-order bit, p being odd.

If it is sufficient to know $\pm P$ up to sign, one can ignore $x_P \bmod 2$ and consider only the little-endian 255-bit integer y_P, setting the bit 255 arbitrarily, ignoring its previously defined value, or clearing it.

A.3.8. Private key for Ed25519. A *private key* for Ed25519 is just an arbitrary 256-bit string k. A *secret exponent* a and *secret salt* k'' are derived from k by first computing $\text{SHA512}(k)$, and then taking the first 256 bits of this SHA512 as the little-endian representation of a (but with bits 255, 2, 1, and 0 cleared, and bit 254 set); the last 256 bits of $\text{SHA512}(k)$ then constitute k''.

This is essentially the same procedure as described in **A.2.4**, but with Curve25519 replaced by the birationally equivalent curve Ed25519. (In fact, it is the other way around: this procedure is standard for Ed25519-based elliptic curve cryptography, and TON extends the procedure to Curve25519.)

A.3.9. Public key for Ed25519. A *public key* corresponding to a private key k for Ed25519 is the standard representation (cf. **A.3.7**) of the point $A = [a]G'$, where a is the secret exponent (cf. **A.3.8**) defined by the private key k.

Notice that $\phi(A)$ is the public key for Curve25519 defined by the same private key k according to **A.2.4** and (45). In this way, we can convert public keys for Ed25519 into corresponding public keys for Curve25519, and vice versa. Private keys do not need to be transformed at all.

A.3.10. Cryptographic Ed25519-signatures. If a message (octet string) M needs to be signed by a private key k defining secret exponent a and secret salt k'', the following computations are performed:

- $r := \text{SHA512}(k''|M)$, interpreted as a little-endian 512-bit integer. Here $s|t$ denotes the concatenation of octet strings s and t.
- $R := [r]G'$ is a point on Ed25519.
- $\tilde{R}$ is the standard representation (cf. **A.3.7**) of point R as a 32-octet string.
- $s := r + a \cdot \text{SHA512}(\tilde{R}|\tilde{A}|M) \bmod \ell$, encoded as a little-endian 256-bit integer. Here $\tilde{A}$ is the standard representation of point $A = [a]G'$, the public key corresponding to k.

The (Schnorr) signature is a 64-octet string $(\tilde{R}, s)$, consisting of the standard representation of the point R and of the 256-bit integer s.

A.3.11. Checking Ed25519-signatures. In order to verify signature $(\tilde{R}, s)$ of a message M, supposedly made by the owner of the private key k corresponding to a known public key A, the following steps are performed:

- Points $[s]G'$ and $R + [\text{SHA512}(\tilde{R}|\tilde{A}|M)]A$ of Ed25519 are computed.
- If these two points coincide, the signature is correct.